Giggle:
A Framework for Constructing Scalable Replica Location Services

Ian Foster Adriana Iamnitchi Matei Ripeanu
Department of Computer Science, University of Chicago, Chicago, IL 60637

Ann Chervenak Ewa Deelman Carl Kesselman
Information Sciences Institute, University of Southern California, Marina del Rey, CA 90292

Wolfgang Hoschek Peter Kunszt Heinz Stockinger Kurt Stockinger
CERN, European Organization for Nuclear Research, CH-1211 Geneva 23, Switzerland

Brian Tierney
Lawrence Berkeley National Laboratory

Abstract

Within high-performance, large-scale wide area computing environments, data replication provides an important mechanism for managing data locality while increasing the reliability of access to critical data. For replicas to be of general use, it is important to have robust and scalable methods for identifying the storage systems on which specific replicas are located. We propose a general framework for creating replica location services, which we call the GIGa-scale Global Location Engine (Giggle). This framework defines a parameterized set of basic mechanisms than can be instantiated in various ways to create a wide range of different replica location services. By adjusting the system parameters, we can trade off reliability, storage and communication overhead, and update and access costs. We present requirements for replica location services, define the behavior of the framework's fundamental components, and use examples to show how the framework can be used to construct replica location services with different properties.

1 Introduction

In a wide area computing system, it may be desirable to create remote read-only copies (*replicas*) of data elements (*files*)—for example, to reduce access latency, increase robustness, or increase the probability that a file can be found associated with idle computing capacity. A system that includes such replicas requires a mechanism for locating them.

We thus define the *replica location problem*: given a unique *logical* identifier for desired data content, determine the *physical* locations of one or more copies of this content. We further define a *replica location service (RLS)* as a component that maintains and provides access to information about the physical locations of copies.

We are interested in developing an RLS that addresses scalability, reliability, and security concerns that arise in large-scale distributed systems, where we may have tens of millions of data items, tens or hundreds of millions of replicas, relatively high update rates, hundreds or perhaps many thousands of replica sites, and a need for high reliability and strong security. In this paper, we motivate these requirements; describe an architectural framework, which we name Giggle (for GIGa-scale Global Location Engine), within which a range of RLSs can be defined; and define instantiations of this framework that we assert meet the identified requirements. We are prototyping several of these instantiations with the goal of validating this assertion in experimental settings.

Replica location is a special case of service discovery, a concept that has been studied extensively [9, 10, 18, 20, 25, 27, 31, 34, 39]. (One can also think of replica location information as a catalog, as metadata, or as information maintained by a distributed file system [1, 22, 32], although we argue against these perspectives.) However, replica location information has special properties that can be exploited to develop specialized structures with better performance characteristics than a generic discovery service. In particular, while replicas may be created and destroyed frequently, an RLS may not need to maintain strictly consistent information about currently existing replicas, since the cost penalty associated with bad information about replicas is often moderate. (By contrast, a *strictly consistent* RLS must *always* return a *complete and accurate* list of copies of the specified content—something that can be expensive or impossible to achieve in a Grid environment [13].) In such environments, resources that are distributed in a wide area environment and under the control of various organizations can be used in a coordinated fashion.

Recognizing that different applications may need to operate at different scales, have different resources, and have different tolerances to inconsistent replica location information, we define a flexible RLS framework, called Giggle (for GIGa-scale Global Location Engine), that allows users to make tradeoffs among consistency, space overhead, reliability, update costs, and query costs by varying six simple system parameters. The Giggle framework is structured in terms of five basic mechanisms. At each replica site, a local name resolution catalog is maintained to provide for *representation of local state*. Replica sites propagate information to one or more replica location index nodes, which provide for *unreliable representation of global state*. Information is communicated from replica sites to index nodes using *soft state techniques*, simplifying the maintenance of global state. This information can be *compressed* prior to communication, potentially reducing communication costs at the expense of some reduction in global state accuracy. Finally, a *membership protocol* allows replica sites and index nodes to locate each other and configure themselves appropriately. In the Giggle framework, six system parameters control the operation of these mechanisms. We note that what constitutes a "replica site" is application-specific. Some applications may consider a single storage system to be a replica site, whereas others might view a replica site as a collection of storage systems.

Initial performance experiments suggest that RLS implementations constructed within this framework can provide significantly better scalability and reliability than previous replica location service designs. For example, we have obtained initial results from a nonoptimized Python prototype of our local catalog that uses Bloom filter encodings [3] to compress information sent by local catalogs. Queries against a local catalog of one million files from a single client on a LAN achieved over 300 queries per second (without authentication); with multiple clients, throughput to this catalog peaked at 3000 queries per second. While we still have to evaluate the cost of global state maintenance and authentication, these initial results are encouraging.

2 Requirements

As large "Data Grids" [6, 21] are still a new phenomenon, we lack comprehensive information concerning scale, size, access patterns, performance needs, and other requirements for an effective RLS. However, we do have information from two communities, high energy physics (HEP) and climate simulation, that we review here.

In the remainder of this paper, we use the following terminology. A *logical file name (LFN)* is the unique logical identifier for desired data content. The RLS must identify zero or more physical copies of the content specified by an LFN. Each physical copy is specified by a unique *physical file name (PFN)* such as a GridFTP [2] URL, which specifies its location on a storage system. We also emphasize that the concept of a unique logical identifier for a desired data content is applicable only in the context of a virtual organization (VO) [15] that brings together users and resources in the pursuit of common goals. .

2.1 Read-Only vs. Mutable Data

One major access modality within a Data Grid, and the one that particularly motivates our interest in replica location services, is access to read-only data. It is frequently the case that data of interest to a large community are prepared, annotated, and then published prior to delivery to the community; after this act of publication, they are immutable. In this paper, we assume that data are immutable except in the context of replica creation and removal. In some applications, mutable files can be supported by versioning. In other more general situations, we expect Data Grids to provide a separate mechanism to coordinate updates to replicated data rather than forcing a consistency requirement on the replica location service.

2.2 Scale, Size, and Performance

In both HEP and climate modeling applications, it is assumed that replicas will be maintained at specialized centers that dedicate significant storage and computing for this purpose. The number of such *replica sites* is envisioned to be at most a few hundred, with a smaller number of sites having significantly more storage and computing than others. The number of users is significantly greater, certainly numbering in the tens of thousands. In principle, computers and storage provided by individual users could be considered replica sites as well, in which case the scale of the system could start to approach that of peer-to-peer networks [26]. However, this mode of use is not currently of interest to our target communities and so we do not consider it here. We plan to address this topic in future work.

The RLS must scale to support large numbers of logical files and replicas and fairly high query and update rates. Approximations of these requirements for the HEP experiments, currently the most demanding application domain, are as follows:

- Support up to 200 replica sites
- 50 million logical files catalogued in total
- 500 million physical files (replicas) in total
- 20 million physical files at a (large) site
- Average query response times of 10 milliseconds
- Maximum query response times of 5 seconds
- Maximum query rates of 100 to1000 per second (burst)

- Maximum update/insertion rate of 50 to 200 per second (burst)

We are particularly uncertain about the maximum query and update rates, and can imagine situations in which they could be at least order of magnitude higher.

2.3 Security

Data Grid administrators and users wish to control who can create, delete, read, and modify data content. For the most part, the replica access control problem is no different than the general Grid security problem. Replica sites need to verify that users are authorized to perform requested operations, while users want assurances that results of RLS queries are valid. Sites and users both may make authorization decisions on the basis of either identity or capabilities provided by approved authorizing entities.

We identify two security requirements for replica information:

- *Privacy*: Some data may be private, and knowledge of its existence, location, and content must be controlled.
- *Integrity*: We should prevent an adversary from tampering with replica location results returned by RLS queries and the contents of the replicas themselves.

In general, we can partition security concerns between the RLS and the storage systems that maintain replicas: the RLS is most concerned with protecting the privacy and integrity of knowledge about the existence and location of data, while individual storage systems protect the privacy and integrity of data contents.

2.4 Consistency

In a Grid environment, where local sites may delete replicas or become disconnected without warning, it is impossible to provide a completely consistent view of replica location. Depending on when and from where information about replica location is requested, different answers may be returned. Fortunately, as we noted in the introduction, an RLS need not provide such a completely consistent view. The reason is that replica information is concerned not with logical function but with performance. For example, in an extreme case where we had no RLS at all, and every access to a logical file had to go to some original source on slow tertiary storage device, our applications would still run correctly, albeit inefficiently. Hence, if an RLS sometimes returns only a subset of all extant replicas, or returns a list of replicas that includes “false positives” (i.e., putative replicas that do not in fact exist), the requesting client may execute less efficiently, but will not execute incorrectly.

This is not to say that consistency is not important. The performance impact of incomplete information can be substantial. For example, consider a situation in which two replicas of a large file exists, one on an idle uniprocessor machine and the other on an idle 100-processor machine. An RLS query that identifies only the first of these two locations might result in an analysis operation taking 100 times longer. Thus, when discussing RLS designs, what is important is not achieving absolute consistency but with *enabling and making appropriate tradeoffs between the cost of inconsistency and RLS implementation and operation costs*.

2.5 Reliability

Data Grid environments are subject to a wide range of failures. In designing a RLS, we identify two general requirements with respect to reliability:

- *No single point of failure*. The RLS structure should be such that no one site, if it fails or becomes inaccessible, can render the entire service inoperable.
- *Decoupling of local and global state*. Failure or inaccessibility of remote RLS components should not affect local access to local replicas.

3 The Giggle Replica Location Service Framework

We now describe the RLS framework that we have designed to meet the requirements listed above. We first summarize the five essential elements of the framework and then describe each element in more detail; in the next section, we describe three different instantiations.

- *Independent local state*. Each replica site maintains a Local Replica Catalog (LRC) used to keep track of the replicas located at a site by recording the LFNs and corresponding-PFN for each such replica. The techniques used to implement LRCs are not specified in our framework, but it is generally desirable that the LRC provide a consistent and reliable representation of local replica state, so that LRC queries always return correct information about replicas at its site.
- *Unreliable global state*. Additional index state is maintained at a set of Replica Location Indices (RLIs), which are responsible for processing queries concerning location of replicas. The number of RLIs and the distribution of index information to those RLIs are determined by four parameters, G, P_L, P_R, and R, described in Section 3.2. RLIs may, but need not, be located at replica sites. RLIs are not required to maintain consistent information about global replica state: a query to an RLI may or may not return complete or accurate information.
- *Soft state maintenance of RLI state*. We propagate information from LRCs to RLIs using soft state techniques [5, 24], as used in various Internet protocols [7, 38]: i.e., information that times out and must be periodically refreshed. A state propagation algorithm, S, is used to determine what information is propagated when: e.g., recent updates and/or all state periodically.
- *Compression of state updates*. A fifth parameter, C, can be used to identify a compression scheme to be applied to LRC state before updates are propagated to RLIs, in order to reduce network traffic and the size of RLIs.
- *Membership service*. A membership service is defined for locating participating replica sites and RLIs.

We assume that RLS users may be distributed anywhere in the Data Grid and that they may query any LRC (for information about replicas located at that specific replica site) or RLI (for information about replicas located at a number of sites).

Within this framework, then, an RLS design may be specified in terms of six parameters, G, P_L, P_R, R, S, and C. We show later how varying these parameters can affect RLS properties.

3.1 Local Replica Catalogs

A *local replica catalog* (LRC) maintains information about replicas at a single replica site. It implements a mapping between arbitrary LFNs and physical file locations (PFNs) on the associated storage system(s). When replicas are created or deleted on the storage system, logical-to-physical mappings for those files should be added and deleted, respectively, within the LRC.

The role of a LRC is to encapsulate essentially *local* policy issues of file naming and placement within a single location or site. For this reason, we intend that LRCs operated under the same policy as the storage systems for which they maintain information and will typically be placed "close" to their corresponding storage system.

An LRC must meet the following requirements:

- *Contents*. It must maintain a mapping between arbitrary LFNs and the PFNs associated with those LFNs on its storage system(s).
- *Queries*. It must respond to the following queries:
    - *What PFN(s) are associated with a specified LFN*?
    - *What LFN is associated with a specified PFN*?
- *Local integrity*. The mechanisms used to update LRC contents are not defined by our framework and may well vary depending on the number, type, and location of storage systems associated with the LRC. It is the responsibility of the LRC to coordinate the name map with the contents of the storage system in an implementation-specific manner. It is generally desirable that the addition or deletion of a replica and the corresponding LRC update occur atomically. An LRC may well use techniques such as backups, replication, and/or checkpoints and update logs to increase reliability. We also assume that the storage system performs the desired level of data corruption detection and recovery.
- *Access control*. Information within the LRC may be subject to access control, and as such must support authentication and authorization mechanisms when processing remote requests. Requests come directly from remote RLS users and so access control must take place within the context of a Grid-based identity. This typically means that an LRC must support Grid Security Infrastructure (GSI) [14] mechanisms for global authentication, integrity, and confidentiality; and Community Authorization Server (CAS) capabilities for authorization.
- *State propagation*. The LRC must periodically send information about its state using a state propagation algorithm, *S*, to RLI(s), as discussed in the next two subsections.

We note that while the only information required in an LRC to meet the requirements above is a set of LFN and PFN fields, a particular RLS implementation might also use the LRC to store additional information, such as file size and information about the logical collection or virtual organization to which the logical file belongs.

Additional requirements can stem from the adoption of the virtual organization (VO) model for Grid computing, where it is desired that VOs coexist and resources are shared between VOs without affecting each other. In this context, multiple LRCs might be associated with a given storage system, each for a different VO. The LRCs would then interact with VO-specific RLIs. This approach also provides for the greatest freedom for VOs to configure the RLS in a way that is most appropriate for their needs. (Various possibilities are shown in Section 4.) Another approach is to have VOs sharing the same storage systems to have additional services, which customize the views of the LRC to a given VO. In this case, the LRCs have to support multiple protocols, implement compression algorithms specific to a VO and send messages to RLIs belonging to a given VO.

3.2 The Global Replica Index

While the various LRCs provide, collectively, a complete and globally consistent record of all extant replicas, they do not directly support user queries of the form “locate replicas for LFN X.” An additional *global index structure* is required to support these queries. In abstract terms, this index structure should allow users to pose queries against a set of (LFN,replica-site) index entries, returning (ideally) a complete and accurate set of successful matches.

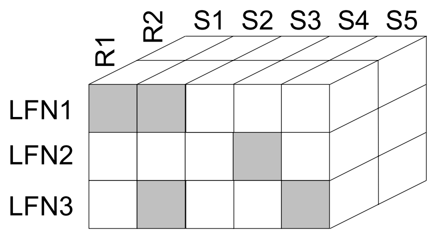

Figure 1: Giggle replica index space (see text for details)

The Giggle framework structures this index as a set of one or more Global Replica Index Nodes (RLIs), each of which contains some number of (LFN,replica-site) index entries. An RLS client that knows how (LFN,replica-site) entries are distributed to RLIs can then determine the RLI(s) to which a particular query can be directed. A variety of index structures can be defined with different performance characteristics, simply by varying the number of RLIs, the amount of redundancy supported in the set of (LFN,replica-site) entries, and the distribution of those index entries to RLIs.

Table 1: The six parameters used to characterize Giggle RLS structures, and some examples of possible values and their implications. See text for more details.

G	**The number of RLIs**	
	G=1	A centralized, nonredundant or partitioned index
	G>1	An index that includes partitioning and/or redundancy
	$G \geq N$	A highly decentralized index with much partitioning and/or redundancy
P_L	**The function used to partition the LFN name space—or ϕ if no partition**	
	$P_L = \phi$	No partitioning by LFN. The RLI(s) must have sufficient storage to record information about all LFNs, a potentially large number
	P_L=hash	“Random” partitioning. Good load balance, perhaps poor locality
	P_L=coll	Partitioning on collection name. Perhaps poor load balance, good locality
P_R	**The function used to partition the replica site name space—or ϕ if no partition**	
	$P_R = \phi$	No partitioning by site name. Indexes have entries for every replica of every LFN they are responsible for. Potentially high storage requirements
	P_R=IP	Partition by domain name or similar. Potential geographic locality
R	**The degree of redundancy in the index space**	
	R=1	No redundancy: each replica is indexed by only one RLI

	R=G>1	Full index of all replicas at each RLI. Implies no partitioning, much redundancy/space overhead
	1<R<G	Each replica indexed at multiple RLIs. Less redundancy/space overhead
C	**The function used to compress LRC information prior to sending—or ϕ if none**	
	C= ϕ	No compression: RLIs receives full LFN/site information
	C=bloom	RLIs receive summaries with accuracy determined by Bloom parameters
	C=coll	RLIs receive summaries based on collection distribution
S	**The function used to determine what LRC information to send when**	
	S=full	Periodically send entire state (perhaps compressed) to relevant RLIs
	S=partial	In addition, send periodic summaries of updates, at a higher frequency.

Abstractly, we can consider the set of index entries as a cube, with the three dimensions being the LFN name space, the set of replica sites, and the degree of redundancy in our index space. For example, Figure 1 shows a cube with three LFNs, five replica sites, and redundancy of factor 2. The shaded squares represent five current replicas. Thus, we have a total of 10 tuples to distribute: two each of (LFN1,S1), (LFN1,S2), (LFN2,S4), (LFN3,S2), and (LFN3,S5).

With some loss of generality, we can then characterize a wide range of global index structures in terms of six parameters (G, P_L, P_R, R, S, C). As summarized in Table 1, four parameters (G, P_L, P_R, R) describe the distribution of replica information to RLIs and two define how information is communicated from LRCs (S, C). The parameter G specifies the total number of RLIs in the replica location service. P_L determines whether there is any partitioning of information sent to the RLIs based on the logical file name space; two proposed partitioning schemes are a hash function on the logical file name or partitioning of index information based on logical collections that group together related logical files. The parameter P_R indicates whether there is any partitioning of the replica sites so that a particular RLI receives state updates from only a portion of the total replica sites. R indicates how many copies of each RLI are maintained in the replica location service. The soft state algorithm S indicates the type of updates sent from LRCs to RLIs. Finally, the parameter C indicates whether a compression scheme is used to reduce the size of soft state updates; proposed compression schemes include using Bloom filters [3, 12] to summarize the contents of an LRC or restricting soft state updates to include information only about logical collections stored at a particular replica site.

Note that in an RLS implementation that includes redundant indices (R>1), we do not assume consistency between the redundant RLIs. We rely on the soft-state protocols to maintain similar information in the different RLIs.

3.2.1 Global Index Structure Options

Different global index structures can have radically different performance characteristics, supporting different tradeoffs among space overheads associated with RLIs, the amount of communication required to maintain the index structure, and the robustness of the index structure in the face of failures. Consider first three cases that do not partition the index information:

- If G=1, then we have a centralized global index. This structure is easy to maintain (we simply direct all LRC updates to the central location) and query, but is not robust to failure or scalable, and has high storage costs at the central site.

- If $G>1$, $R=G$ and $P_L=P_R=\phi$, then we have a full index replicated at G sites. All LRC updates must be sent to all G sites. Storage and communication costs increase with G. Increasing redundancy offers load balancing for queries and resilience to failures.
- If $G=N$ (the number of replica sites), $R=G$, and $P_L=P_R=\phi$, then we have a full index replicated at every replica site. This structure is more expensive to maintain (we must direct all LRC updates to every replica site) and has high storage costs, but is easy to query and is highly robust to failure.

Next, consider four cases that partition the global index and do not include redundancy of index information ($G>1$ and $R=1$). We can partition the set of index entries in various ways to reduce storage and update communication costs (by a factor of G), as illustrated in **Figure 2**.

a. P_L=hash(LFN)mod G and $P_R=\phi$: we partition the LFN name space based on LFN name. Each RLI contains information about all replicas of a subset of all LFNs.

b. $P_L=\phi$ and P_R=some-fn(ip_domain(replica-site)): we partition the replica site name space based on some notion of geographic locality. Each RLI contains information about some replicas for all LFNs.

c. P_L=some-fn(collection(LFN)) and $P_R=\phi$: we partition the LFN name space by collection name. Each RLI contains information about all replicas of a subset of all LFNs, but this time grouped by some "collection" notion.

d. $P_L\neq\phi$, $P_R\neq\phi$: we partition in two dimensions.

Note that partitioning does not improve resilience as we still have a single point of failure for any specific query. Thus, to achieve application requirements for index performance, reliability, and storage and communication costs, many practical RLS implementations will combine redundancy and partitioning.

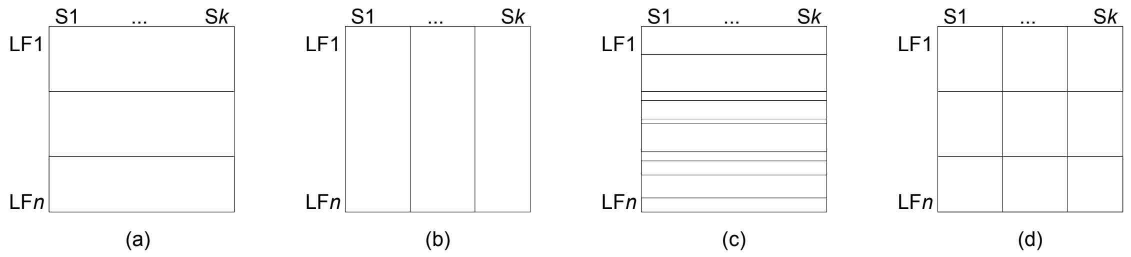

Figure 2: Four different partitionings of the index entry name space, each described in the text. LF1..LF*n* are n logical file names; S1..S*k* name *k* replica sites.

3.2.2 A General-Purpose Hybrid Strategy

One simple RLS design that would perform well in many situations implements a moderate number of RLIs (G= the number of replica sites); has a moderate degree of redundancy ($G>>R>1$); and partitions LFN name space using a simple hash function on LFN name ($P_L\neq\phi$, $P_R=\phi$). As illustrated in Figure 3, this situation has the following properties:

- A client can direct a query for an LFN to any one of the R distinct RLIs that contains the relevant index information. This redundancy provides fault tolerance and load balancing.
- A replica site (LRC) updates the global index structure by sending a new or modified index entry to each of the R distinct RLIs responsible for each LFN to be updated. This scheme imposes moderate communication requirements for updates.

- Storage demands associated with RLIs are moderate, only a factor of *R* greater than the minimum possible.

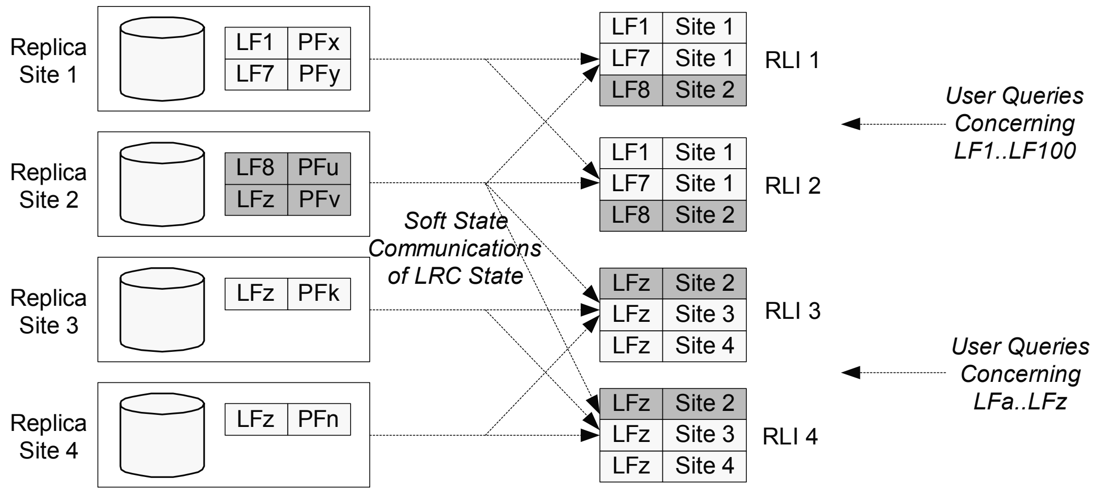

Figure 3: In this schematic, we see four replica sites, to the left, and four RLIs, to the right. Here, we have *G*=4, *R*=2, and P_L is such that replica index entries for numerical LFNs are replicated redundantly at RLI 1 and 2, while index entries for alphabetical LFNs are replicated at RLI 3 and 4. Shading is used to show the information propagated by replica site 2.

3.2.3 Indexing with Redistribution

If the set of RLIs changes over time, then in addition to a membership protocol that keeps track of these changes, we need to be able to redistribute LFNs or replica sites to RLIs. If the number of RLIs is relatively large, then we can address redistribution issues by using consistent hashing [19] as our partitioning algorithm (as used, for example, in Chord [31] and Pastry [11]). Briefly, we partition LFNs or replica sites as follows:

- RLI identifiers are passed through a hash function that returns a value in the range 0..360 (on a circle)
- LFNs (if $P_L \neq \phi$) or replica sites (if $P_R \neq \phi$) go through the same hash function and are associated with the closest RLI clockwise.

Then, when an LRC discovers that an RLI has departed (as determined, for example, via the membership service described in Section 3.6), it sends its soft-state update messages to the next RLI clockwise. Similarly, an RLI that joins the system will take on part of the load of a neighboring RLI.

3.3 Soft State Mechanisms and Relaxed Consistency

We next discuss how LRC updates are propagated to the relevant RLI(s). If we wanted strong consistency, these operations would have to be atomic transactions—although even then we could not prevent network partitions and site failures from rendering the global index inconsistent. However, as strong consistency is not required, we can instead use a *soft state* protocol [5, 7, 24, 38] to send LRC state information to relevant RLIs, which then incorporate this information into their indices.

Soft state is information that times out and must be periodically refreshed. There are two main advantages to soft state mechanisms here. First, stale information is removed implicitly, via time outs, rather than via explicit delete operations. Hence, removal of data associated with failed or inaccessible replica sites can occur automatically. Note that soft state does not eliminate the need to be prepared for a "false positive," as an LRC (or storage system) may fail between the time that an RLI is consulted and the time that the LRC is checked. The second advantage to the use of soft state to populate RLIs is that global index information need not be persistent, as it can be reconstructed after RLI failures using the periodic soft state updates from LRCs.

The use of soft state mechanisms is compatible with our relaxed consistency model for RLS information. RLIs are not required to be strictly consistent with the contents of storage systems and LRCs; temporary delays are allowed in propagating soft state updates from LRCs to RLIs.

Various soft state update strategies can be defined, with different performance characteristics. A simple scheme sends the entire LRC state periodically. More sophisticated schemes might interpolate state updates. Communication frequency can be varied according to network load [29]. We define as *S* the soft state communication algorithm used.

3.4 Requirements for Replica Location Indices

Based on the redundancy, partitioning, and soft state mechanisms we have described for the Giggle framework, we can now summarize requirements that must be met by a global replica index node (RLI).

- *Secure remote access.* AN RLI must support remote access and must implement the Grid Security Infrastructure (GSI) [14] for authentication, integrity and confidentiality; accept Community Authorization Server (CAS) [23] capabilities; and implement local access control over its contents.
- *State propagation.* It must accept periodic inputs from LRCs describing their state. If the RLI already contains an LFN entry associated with the LRC, then the existing information is updated or replaced. Otherwise, the index node creates a new entry.
- *Queries.* It must respond to queries asking for replicas associated with a specified LFN by returning information about that LFN or an indication that the LFN is not present in the index.
- *Soft state.* An RLI must implement time outs of information stored in the index. If an RLI entry associated with an LRC has not received updated state information from the LRC in the specified time out interval, the RLI must discard that entry.
- *Failure recovery.* An RLI must contain no persistent replica state information. That is, it must be possible to recreate its contents following RLI failure using only soft state updates from LRCs.

In [9], we make the case for hierarchal indexes as a means of managing information locality and presenting multiple views of the same underlying information. The same arguments can be made within the context of an RLS, as illustrated in Section 4.6. The construction of a hierarchal index adds the requirement that an RLI propagate state using the same soft state protocol used by an LRC.

3.5 Compression

We have not talked in detail about the information that is communicated from an LRC to RLI(s) to update their indices. A simple technique is to communicate the LFNs located at the LRC. This "uncompressed" information allows the RLI(s) to maintain an index that is completely accurate, modulo time delays between when changes occur at the LRC and when updates are processed at the RLI(s). However, we can also compress LFN information in various ways. For example:

- We can compress information using hash digest techniques such as Bloom filters [3, 12].
- We can use structural or semantic information in LFNs, such as logical collection names—user-defined groups of files (i.e., named data sets) that partition the logical file namespace according to application-specific characteristics. In this case, an RLI entry would map logical collection names, rather than LFNs, to replica sites.

The motivation for compression is to reduce overall network traffic and the cost of maintaining RLIs. The benefits of compression need to be evaluated in practical settings.

3.6 Membership Service

The set of replica sites and RLIs associated with an RLS may change over time, in which case we require mechanisms to accommodate these changes. The nature of these mechanisms depends on the partitioning strategy used.

- $P_L=\phi$, $P_R=\phi$ (i.e., there is no partitioning of the index space). Here, things are straightforward. If a new replica site joins, it needs a way of locating the RLI(s) to which it should send replicas. If an RLI joins or leaves, we need a way of notifying the various LRCs that the RLI(s) to which they send information has changed. Both issues can be addressed via the *membership service* presented below.
- $P_L \neq\phi$, $P_R=\phi$ (i.e., we partition by LFN but not by replica site). Here, things are a little more complex. If a new replica site joins, we use the membership service, as before, to locate the RLI(s) to which it should send replicas. If an RLI joins or leaves, we might want to repartition LFNs among the new set of RLI(s), depending on the amount of redundancy in our global index. This issue can be addressed via a redistribution mechanism, as discussed in Section 3.2.3.
- $P_L=\phi$, $P_R \neq\phi$ (i.e., we partition by replica site but not by LFN). If a new replica site joins, we use, as before, the membership service to locate the RLI(s) to which it should send replicas. We might also want to repartition replica sites among RLIs, which we can achieve using our redistribution mechanism. If an RLI joins or leaves, we might want to use our redistribution mechanism to repartition replica sites among the new set of RLI(s), depending on the amount of redundancy in the global index.

The *membership service* keeps track of currently active RLI(s) and responds to (a) requests for a list of active RLIs in a given VO and (b) requests for asynchronous notification of changes in the list of active RLIs. Note that our use of a soft state protocol means that RLIs learn automatically of changes in replica site status.

Various membership service implementations are possible. Given the scale assumptions of Section 2.2, it may be reasonable to use a centralized server (perhaps with multiple representatives) and a soft state registration protocol [9]. In larger systems, a hierarchical organization could be used. The grid information service architecture proposed in [9] meets these requirements, and it would appear that the MDS-2 implementation of this architecture could be used as an effective RLI membership service.

An alternative, more complex but also more scalable implementation strategy for the membership service would use a decentralized, soft state membership protocol based on gossip messages exchanged by RLIs [16]. Such a protocol can guarantee, with high probability, the consistency of the membership information. (Consistent membership information is impossible to achieve [4] in asynchronous systems if failures are possible.) For system bootstrap, we could use a mechanism based on DNS [20] (similar to that used by Akamai and other content distribution networks): a new RLI launches a DNS request for a known address (say cms.rls.net) that is answered with the location of the best (in terms of network latency, estimated throughput, etc.) node that is already a member of the RLI group.

We note that membership information provided by this service can be used not only for the purposes considered here, but also for exhaustive searches: for example, when updating mutable data. Given the RLI membership information, we can imagine various ways to connect RLIs in reliable, search-efficient structures, such as a complete graph for a small number of RLIs, or overlapped trees with efficient flooding mechanisms used to forward requests.

From the client perspective, it is not sufficient to know which RLIs are available. Additional information is required for the client to be able to determine which set of RLIs contains the desired mapping. Since this depends on the partitioning strategy, the parameters used for this partitioning (as described in Section 3.2) need to be available as well. Information about these potentially dynamic parameters needs to be obtained through the use of membership services or via other information services such as MDS-2.

4 Implementation Approaches

Having described our general framework, we now illustrate its application by showing how it can be realized in six different RLS implementations.

4.1 RLS1: A Centralized, Nonredundant Global Index

Probably the simplest global index structure consists of LRCs that send their full, uncompressed state periodically to a single RLI, with no redundancy or partitioning. That is,

$$G=R=1 \text{ (and hence } P_L=P_R=\phi\text{)}, S=\text{all}, C=\phi.$$

Each LRC sends to the RLI information about every LFN for which it contains a mapping; the RLI adds these LFNs to its index along with the associated replica site name. Note that the RLI may contain multiple entries for an LFN if more than one LRC contains mappings for that LFN. The approach is illustrated in Figure 4, which shows a simple configuration involving three LRCs.

A client query to the RLI returns the names of all replica sites known to contain a copy of the specified LFN. The client must next consult the LRCs directly to determine the physical locations of the desired replicas. Since RLIs do not maintain strict consistency with storage systems, the client may then be informed that the desired replica no longer exists at an LRC. Notice that keeping the mappings to physical files in the LRC rather than propagating them to the RLI gives storage system managers the freedom to move or delete files (and update associated LRC entries) at any time, independent of RLI operations.

When a new replica is created on a local storage system, the associated LRC is updated to add a mapping entry. Periodically, each LRC sends its state to the RLI. There is no need to send delete message when replicas are deleted; old index node entries time out and are discarded.

This RLS design is simple but does have some limitations due to the use of a single RLI to maintain the global index. This design introduces a single point of failure (limiting system reliability) and creates a likely performance bottleneck. These limitations are likely to be critical, for example, in the HEP Data Grids, which physicists estimate will eventually contain hundreds of replica sites, approximately 50 million logical files, and 500 million physical files, and will service thousands of client queries per second.

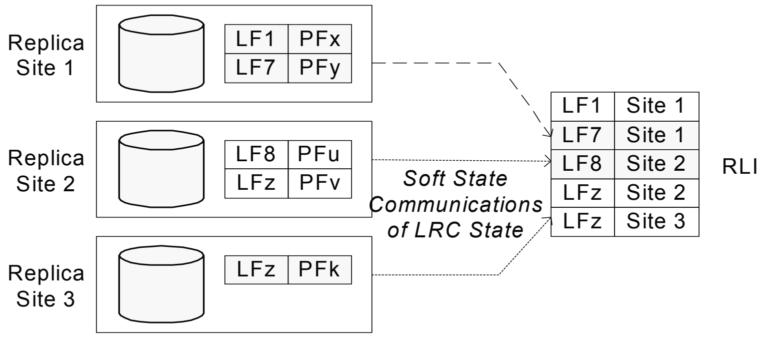

Figure 4: A possible configuration of our RLS1 implementation. A single RLI maintains information about three replica sites.

4.2 RLS2: LFN Partitioning, Redundancy, Bloom Filters

Our second RLS design, illustrated in Figure 5, instantiates our framework rather differently, using the advanced mechanisms described previously to provide improved scalability, reliability and performance:

G>1, R=2, P_L=partition-on-LFN, P_R=ϕ, S=partial, C=Bloom filtering

RLS2 differs from RLS1 in four respects:

- *It includes redundancy*. In Figure 5, there are two copies of each index, which increases total storage requirements but improves reliability, since the RLS can continue to operate after the failure of any index node. Furthermore, since either redundant RLS index node can respond to a client query, the system can provide load balancing. Both index nodes must be updated with any state changes from the LRCs.
- *Index information is partitioned based on the logical file namespace*. In Figure 5, we show two RLI sets (each of size two, due to redundancy), one containing index information for LFN1 through LFN100, and the other for LFNa through LFNz.
- *LRC state information is communicated in compressed form, using Bloom filters* to generate compressed summaries.
- *LRC state information is communicated not only via periodic full-state (compressed) transfers, but also via interleaved partial-state updates.*

Bloom filter techniques perform a series of hash functions on each LFN, generating several values for each LFN that are used to set corresponding bits in a large bit vector. This summary bit vector has several desirable characteristics. First, the bitmaps can be sent fairly efficiently from LRCs to RLIs. Second, we can consult the bitmap to determine with high probability whether a particular LFN is stored on the LRC: we perform the same series of hash functions for the desired LFN and compare the hash values with the bit map. If any corresponding bit is not set, then no mapping for the LFN exists in the LRC. If all corresponding bits are set, there is a high probability that a mapping does exists. (There is a small probability of a “false positive” due to collisions on hash values.)

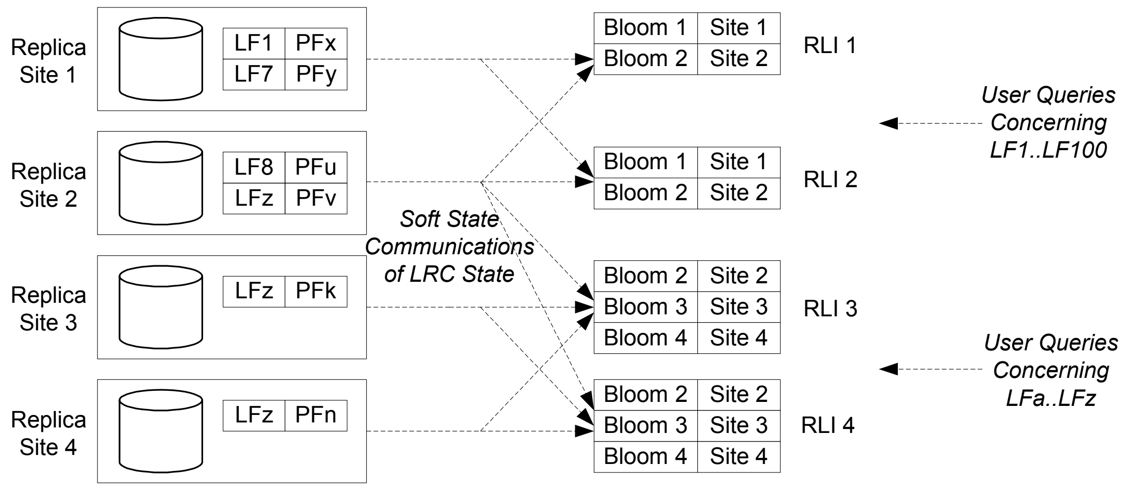

Figure 5: A possible configuration of our RLS2 implementation. See text for details.

As shown in Figure 5, each RLI contains a Bloom filter summary for each LRC, summarizing the LFNs at that site for which the RLI is responsible. (As an optimization, null Bloom filters are omitted.) Each LRC sends these Bloom filter summaries only to those index nodes that are responsible for indexing its LFNs. Thus, since replica site 1 contains only mappings for numeric LFNs, it sends summaries only RLIs 1 and 2. Since sites 3 and 4 contain only mappings for alphabetic LFNs, they sends their summaries only to RLIs 3 and 4. LRC2 sends summarizes to all RLIs. Thus, partitioning the LFN space reduces communication costs as well as storage space requirements on RLIs.

When we want to locate replicas for a particular LFN, we must first determine which RLIs to contact. (Here, “we” may be the client or alternatively the LRC, depending on where we want to place the logic for locating replicas.) Thus, we must be aware of the partitioning function P_L (this can be done with the use of membership services described in Section 3.6). We then contact the RLI, which computes the Bloom filter of the LFN, checks the corresponding bits in all its bit map entries, and returns the names of those LRCs for which all bits are set. We then consult the LRCs to obtain the physical file locations. As in RLS1, it is possible that the user may not find a mapping for an LFN on a particular LRC, either because the file has been deleted or because there was a collision on hash values in the Bloom filter summary.

When replicas are created or deleted on a local storage system, the LRC is updated. Periodically, the LRC re-calculates the Bloom filter summary for the node and sends it to the appropriate index nodes, which use the new soft state information to replace old entries. Both creations and deletions of replicas are reflected in the new LRC state summaries. As an optimization, an LRC may also send summaries of recent updates rather than complete state.

Note that the RLIs in this example store mappings from Bloom filter summaries to replica sites, while in the previous example they contained mappings from LFNs to replica sites. Since the type of soft state updates sent by LRCs differs in these two examples, the implementation of LRCs is not fully decoupled from the RLI type. One question that arises is whether multiple virtual organizations (VOs) that share storage systems may share LRCs, since the VOs may have different requirements for the type of state information sent to RLIs. One option is to maintain separate LRCs for the different VOs, so that each VO can determine its preferred information propagation scheme without imposing it on other VOs using the same storage systems. Alternatively, a single LRC could speak multiple protocols and send different state updates depending on the RLI with which it communicates. The advantage of the latter scheme is that it allows the storage system to keep LRCs at a desired level of consistency without worrying about the VO space structure.

The combination of partitioning, redundancy and compression techniques used in this design offers better scalability, reliability and performance than RLS1. Storage and communication requirements are reduced using compression and partitioning techniques, improving scalability for large Data Grids. While overall storage and communication requirements increase with the use of redundant RLIs, this technique provides better reliability as well as load balancing for improved performance.

4.3 RLS3: Referral Service

An extrapolation of the RLS described above uses "extreme" compression techniques instead of Bloom filters. Let us assume that the only information that an LRC sends to the RLI is that it has files in the partition maintained by the RLI. Here the compression technique compresses the LFN/PFN information down to zero.

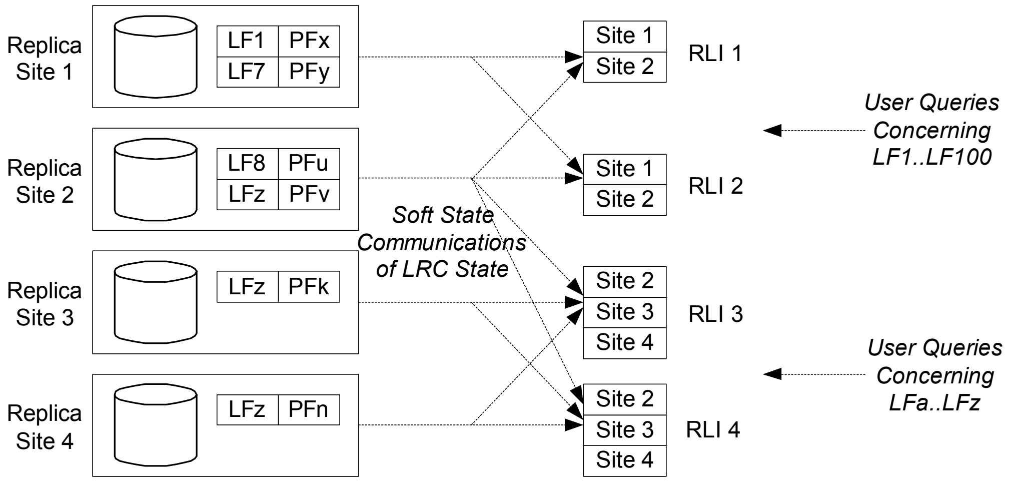

Figure 6: Example configuration for RLS3 referral service

In that case, an RLI would effectively "refer" clients to all the storage systems for which it performs indexing. The burden of efficiency of this approach relies heavily on the number of RLIs and the partitioning of the logical file namespace. The drawback of this approach is that the client needs to perform queries to possibly many LRCs. However, it has the potential of being efficient in cases where a client has a preference for which resources to use (perhaps based on previous resource access) or has the freedom to dispatch parallel queries and pick the first responsive resource. An additional benefit is that the RLI logic is extremely simple and the amount of information sent in the system is small: proportional to the number of LRCs.

4.4 RLS4: Compression, Partitioning based on Collections

Our fourth RLS design, illustrated in Figure 7, uses our framework to implement both compression and partitioning of the logical file namespace based on domain-specific logical collections. This approach may be advantageous in situations where logical files are grouped into datasets and replication of significant fractions of a data set is common. An example of such a situation occurs in timestepped simulations where the output of the simulation will consist of a set of files, each containing one timestep.

The example we present includes redundancy along with LFN partitioning, so the framework parameters are as follows:

$$G>1,\ R=2,\ P_L=\text{coll},\ P_R=\phi,\ S=\text{full},\ C=\text{collection-named-based}$$

The figure shows four replica sites. To implement compression based on logical collections, we assume that the compression engine has access to mappings between logical filenames and logical collection names. In this example, these mappings are as follows: (LF1, Coll7), (LF7, Coll3), (LF8, Coll3), (LFz, CollX). The compression algorithm notes all logical collections that are mapped by the LRC and sends state updates to the appropriate RLIs.

In addition to compressing soft state updates based on collection information, the global index is also partitioned based on logical collections in this example. Each RLI contains index information about a subset of all logical collections. In the example shown, RLI1 and RLI2 are redundant index nodes for information about logical collections LC1 through LC20. RLI2 and RLI3 are redundant index nodes for information about collections LCA through LCZ.

Based on this partitioning, each LRC sends soft state information about its logical collections to the RLIs responsible for those collections. Since one LRC may contain entries for many logical files belonging to the same logical collection, this design can substantially reduce communication overhead for updates and storage requirements on RLIs compared to other RLS designs by sending only one state update per logical collection.

However, this collection-based compression of LRC state information is lossy, as information about which logical files from a collection are stored on a storage system is not communicated. To determine whether a particular LFN is resident on a storage system, the client must consult the LRC directly. If logical collections are sparsely replicated (i.e., only a small subset of a large collection is typically stored on each storage site), then this scheme may prove inefficient, since the client may need to visit multiple LRCs before finding the desired file (although queries to various LRCs can be executed in parallel). For such sparse replication patterns, the collection-based scheme would perform less well than other designs.

In this scheme, RLI entries associate site pointers with logical collection names rather than logical files. Thus, to query the global index for copies of a particular logical file, the client must know which logical collection holds the desired file as well as the mapping function from the logical collection name to the set of redundant index nodes that contain collection information.

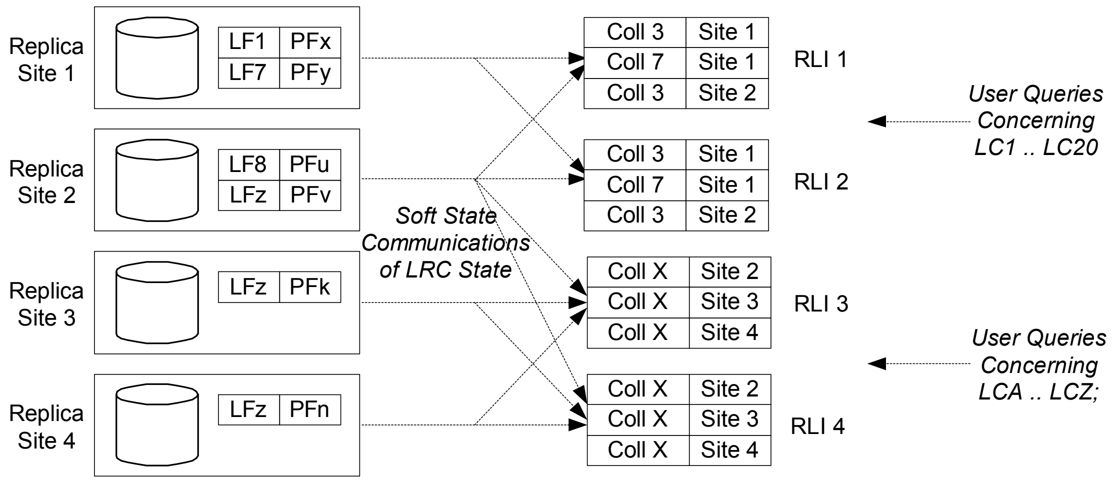

Figure 7: Structure of RLS4, which uses collection names for partitioning function.

When logical files are created or deleted on a storage system, the associated LRC is updated. However, the corresponding RLIs need be updated only when logical collection information changes. For example, consider creation of new replicas on a storage system. If these replicas are of files in a logical collection *Y* that is not currently registered in the associated LRC, then the LRC must eventually send a soft state update to inform the appropriate RLI(s) that the LRC now contains mappings for logical collection *Y*. If additional replicas from logical collection *Y* are later added to the storage system, the soft state information sent to RLIs does not change. Similarly, deletions of logical files are only reflected in soft state updates to RLIs if they correspond to the removal of all files associated with a logical collection. Thus, the collection-based compression scheme can greatly reduce the amount of soft state information propagated from LRCs to RLIs.

Partitioning based on logical collection information has the advantage that it allows partitioning of logical files and RLIs based on domain knowledge of file semantics, file creation and file access patterns. Certain applications will find logical collections useful in organizing their data and in limiting the communication and storage requirements of a replica location service.

4.5 RLS5: Storage System Partitioning, Redundancy, Bloom Filters

Figure 8 shows our fifth RLS design, which uses our framework to implement storage system partitioning. This example also includes redundancy, so the framework parameters are:

$$G>1,\ R=2,\ P_L=\phi,\ P_R=\text{IP},\ S=\text{full},\ C=\text{bloom}$$

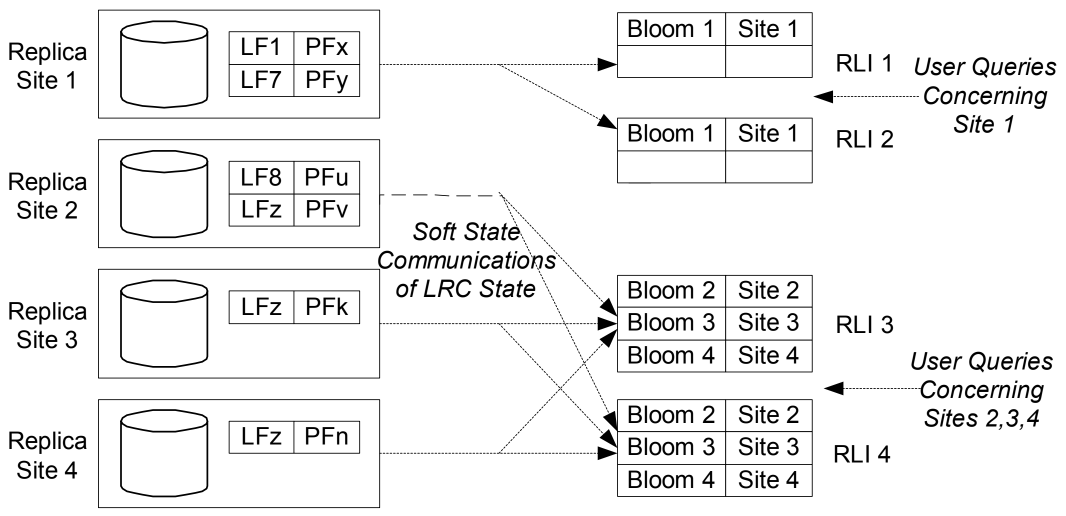

Figure 8: RLS5, an RLS design that uses storage system partitioning.

In this example, RLIs 1 and 2 contain replicated index information for replica site 1, while RLIs 3 and 4 contain replicated information for sites 2, 3 and 4. Storage system partitioning may be useful for Data Grids that are distributed geographically or where storage systems and RLIs are under different administrative control.

4.6 RLS6: A Hierarchical Index

A final example, illustrated in Figure 9, extends RLS2 (partitioning, redundancy, bloom filter compression) to include a second level of RLIs. The figure shows RLIs 1, 2, 3, and 4 sending soft-state updates of their contents to redundant higher-level RLIs 5 and 6. By enabling creation of higher-level indexes on top of any of the partitioning strategies we have discussed, our flexible RLS framework supports a tremendous variety of RLS designs.

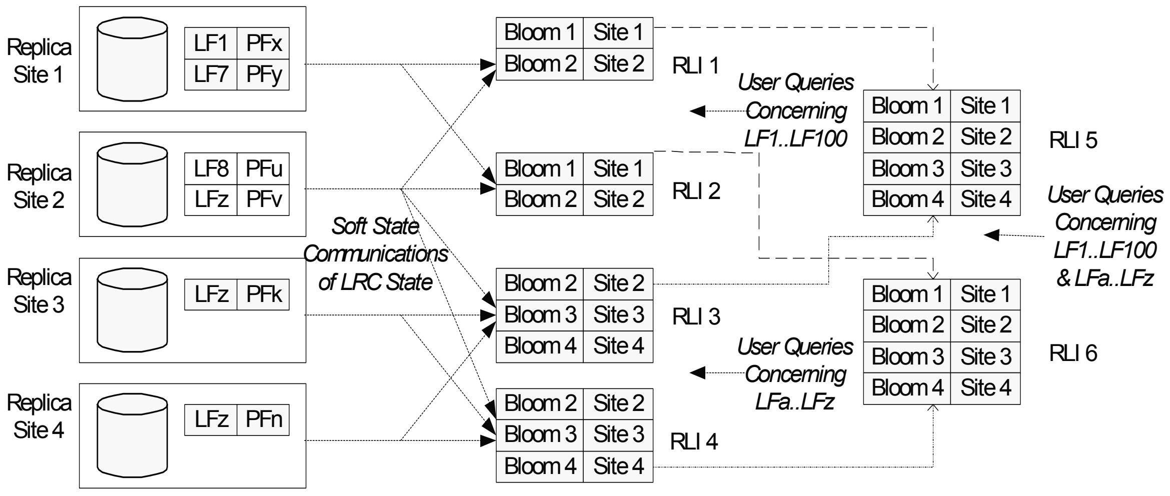

Figure 9: RLS6, an RLS design that incorporates hierarchy.

5 Relaxed Consistency: Discussion

A distinguishing characteristic of the Giggle RLS architecture is that it does not aim for complete consistency. We are aware of a variety of reasons given for why consistency is important. We present these arguments here, and explain why none convince us that consistency should be a requirement for a general-purpose RLS. Note that we do identify a number of areas where consistency is important, but we argue that these cases should be addressed with special-purpose mechanisms.

Metadata must be consistent. We agree that, in general, *metadata* must be consistent. However, we argue that metadata should be maintained via a separate mechanism: we are concerned only with *replica location* information.

*We may not be able to locate any copy of a file that, in fact, exist*s. This situation could arise if, for example, state has been lost and not yet recreated, or if a unique replica has been moved from one location to another, and RLIs have not yet been updated. One approach to this problem would be to repeat the query. Another would be to broadcast a query to all LRCs, although that may prove to be very expensive

We need to be able to locate all copies of mutable files. If files are not read-only, then it can be important to update (or delete) all copies when a designated "master" is updated. But we assume that mutable files are rare and therefore should be treated as a special case, rather than forcing consistency on the entire RLS. For example, we could have a separate *consistent* replica catalog for this purpose or could use broadcast mechanisms to locate all replicas before modifying or deleting a file.

We need to be able to restore the state of replica sites in the event of failure. In this view, the RLS is used to maintain persistent information about the state of a data grid; if a site loses its local data, or a transfer fails, this information can be used to recreate the entire system to its "correct" state. However, we argue that this view puts the cart before the horse: an RLS is intended to communicate information about replicas created by other means, not to drive replica creation. If we want to maintain reliably the status of a series of transfers, then we can use specialized technology designed for that purpose, such as the reliable transfer services being developed in various Data Grid projects. If replicas happen to be deleted, then we can create new replicas using whatever algorithm created them in the past.

6 Related Work

We previously developed a *replica catalog* service [2] designed to provide a consistent view of replica location. Our initial implementation used a centralized catalog and is integrated with replica management tools that reliably create and delete replicas. This service has seen extensive use (e.g., [30]), but has scalability limitations.

Service and resource discovery have been a topic of extensive research. The most relevant to our current work are resource discovery systems in which resources are uniquely identified by an attribute (usually a *name*): CAN [25], Chord [31], Tapestry [39], Gnutella [26], Freenet [8], and Napster. An exception is Ninja's Service Discovery Service [10, 17], in which services, identified by sets of attributes, are given a "name" built as a Bloom filter on a subset of attributes.

CAN [25], Chord [31], and Tapestry [39] build search-efficient indexing structures that provide good scalability and search performance at the increased cost of file insertion and removal. Gnutella [26] does not use indexing mechanisms; its relatively good search performance (as measured in number of hops) is due to intensive network usage. Napster uses a centralized approach: a file index is maintained at a central location, while real data (files) are widely distributed. Freenet [8] includes, in our terminology, both replica management and replica location mechanisms: popular files are replicated closer to users, while the least popular files eventually disappear. Freenet's file location mechanism is built based on usage patterns, using dynamic routing tables. However, Freenet's assumption that non-popular data is unimportant data is not valid for many scientific applications.

In Ninja's Service Discovery Service [17], multiple location servers are organized in a search-efficient hierarchy. Each server is responsible for its local domain and maintains information about existent services in a compressed table based on Bloom filters. Servers propagate their tables upwards in the hierarchy for search efficiency. Scalability is achieved using a lossy aggregation of these tables that preserves the table size independent of the number of services represented.

Cooperative Internet proxy-caches [36] offer similar functionalities to our RLS. A cooperative proxy-cache receives requests for Web pages and is able to locate cached replicas at other collaborating proxies. Hierarchical caching in proxy servers has been extensively analyzed [35, 37]. Two distinct solutions that do not use hierarchies are Summary Cache and the Cache Array Routing Protocol. Summary Cache [12] uses Bloom filters (dubbed "cache summaries") as compact representations for the local set of cached files. Each cache periodically broadcasts its summary to all members of the distributed cache. Using all summaries received, a cache node has a (partially outdated) global image of the set of files stored in the aggregated cache. The Cache Array Routing Protocol [28, 33] uses uniform hash functions to partition the name space and to route requests.

Soft state techniques are used in various Internet services, for example RSVP [7, 38]. The Globus Toolkit's Meta Directory Service (MDS-2) uses soft state concepts to propagate information about the existence and state of Grid resources [9].

7 Summary

We have proposed a solution to the *replica location problem*: given a unique logical identifier for desired data content in a wide area system, determine the physical locations of one or more copies of this content. A *replica location service* (RLS) is a system component that maintains information and answers queries about the physical location of copies. We argue that applications will have different requirements for an RLS depending on their scale, resources, and tolerance for inconsistent replica location information.

We have introduced Giggle, a flexible framework for constructing RLSs that allows users to make tradeoffs among consistency, space overhead, reliability, update costs, and query costs. Giggle is based on five basic mechanisms: representation of local replica state in local replica catalogs (LRCs); distribution of global state among one or more replica location indices (RLIs); the use of soft state techniques to communicate state information from LRCs to RLIs; the optional use of compression to reduce communication and storage overheads for maintaining global state; and a membership protocol that allows LRCs to locate RLIs. The Giggle framework defines six simple system parameters that determine the configuration of a particular RLS.

To demonstrate the flexibility of the Giggle framework, we have presented six different RLS designs. These designs range from a simple single-node centralized global index to a complex hierarchical scheme. By varying the six Giggle system parameters, our framework supports a variety of compression schemes (Bloom filters, logical collections) for soft state information, partitioning strategies (based on logical filenames, logical collections, or storage systems) and replication patterns for global index nodes. The examples demonstrate that the Giggle framework supports a wide variety of RLS designs offering varying levels of scalability, consistency, reliability, and cost.

In conclusion, we note that while the Giggle framework is general enough to encompass a range of different RLS designs, there are certainly alternatives that are not captured and that could be explored in future work. For example, we assume that the topology of the global index structure is known and exploit this fact when distributing updates and queries. An alternative to a fixed topology is peer-to-peer distribution networks [25, 31]. More scalable and adaptive to a highly variable number of index nodes, peer-to-peer distribution networks allow the number of index nodes to vary over time. In these systems, queries are routed among index nodes towards the appropriate node(s). Alternatively, a peer-to-peer network in which information is not indexed by logical file names would provide a more scalable and robust [1], but perhaps less efficient structure. We could also imagine a system that creates indices based on usage patterns, in which each index node is specialized to answer correctly the queries of a group of users and can collaborate with other index nodes to solve unexpected queries [18].

8 Acknowledgements

We are grateful to colleagues for discussions on the topics covered in this paper, in particular (in alphabetical order, with apologies to anybody we've missed): Sridhar Gullapalli, Koen Holtman, Leila Meshkat, Laura Pearlman, Von Welch, and Mike Wilde.

This work was supported in part by the NSF as part of the GriPhyN project and by the DOE as part of the SciDAC program.

9 References

1. Alexandrov, A.D., Ibel, M., Schauser, K.E. and Scheiman, C.J. Extending the Operating System at the User Level: The UFO Global File System. In *1997 Annual Technical Conference on UNIX and Advanced Computing Systems (USENIX'97)*, 1997.
2. Allcock, W., Bester, J., Bresnahan, J., Chervenak, A., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D. and Tuecke, S. Data Management and Transfer in High-Performance Computational Grid Environments. *Parallel Computing*. 2001.
3. Bloom, B. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of ACM*, *13* (7). 422-426. 1970.
4. Chandra, T.D., Hadzilacos, V., Toueg, S. and Charron-Bost, B., On the Impossibility of Group Membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, (New York, USA, 1996), 322--330
5. Chandy, K.M., Rifkin, A. and Schooler, E. Using Announce-Listen with Global Events to Develop Distributed Control Systems. *Concurrency: Practice and Experience*, *10* (11-13). 1021-1027. 1998.
6. Chervenak, A., Foster, I., Kesselman, C., Salisbury, C. and Tuecke, S. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Data Sets. *J. Network and Computer Applications* (23). 187-200. 2001.
7. Clark, D.D., The Design Philosophy of the DARPA Internet Protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, (1988), ACM Press, 106-114
8. Clarke, I., Sandberg, O., Wiley, B. and Hong, T.W., Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *ICSI Workshop on Design Issues in Anonymity and Unobservability*, (1999). http://freenet.sourceforge.net/icsi.ps

9. Czajkowski, K., Fitzgerald, S., Foster, I. and Kesselman, C., Grid Information Services for Distributed Resource Sharing. In *10th IEEE International Symposium on High Performance Distributed Computing*, (2001), IEEE Press, 181-184
10. Czerwinski, S.E., Zhao, B.Y., Hodes, T.D., Joseph, A.D. and Katz, R.H., An Architecture for a Secure Service Discovery Service. In *Mobicom '99*, (1999), ACM Press
11. Druschel, P. and Rowstron, A., PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, (Elmau/Oberbayern, Germany, 2001). http://research.microsoft.com/~antr/PAST/
12. Fan, L., Cao, P., Almeida, J. and Broder, A.Z. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, *8* (3). 281-293. 2000.
13. Foster, I. and Kesselman, C. (eds.). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.
14. Foster, I., Kesselman, C., Tsudik, G. and Tuecke, S. A Security Architecture for Computational Grids. In *ACM Conference on Computers and Security*, 1998, 83-91.
15. Foster, I., Kesselman, C. and Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, *15* (3). 200-222. 2001. www.globus.org/research/papers/anatomy.pdf.
16. Golding, R.A. Weak-consistency group communication and membership *Computer Science*, University of California at Santa Cruz, 1992
17. Hodes, T.D., Czerwinski, S.E., Zhao, B., Joseph, A.D. and Katz, R.H. An Architecture for Secure Wide-Area Service Discovery. *Wireless Networks*. 2001.
18. Iamnitchi, A. and Foster, I., On Fully Decentralized Resource Discovery in Grid Environments. In *International Workshop on Grid Computing*, (2001)
19. Karger, D.R., Lehman, E., Leighton, F.T., Panigrahy, R., Levine, M.S. and Lewin, D., Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Symposium on Theory of Computing*, (1997), ACM, 654-663
20. Mockapetris, P.V. and Dunlap, K., Development of the Domain Name System. In *SIGCOMM*, (1988), ACM, 123-133
21. Moore, R., Baru, C., Marciano, R., Rajasekar, A. and Wan, M. Data-Intensive Computing. In Foster, I. and Kesselman, C. eds. *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999, 105-129.
22. Neuman, B.C. The Prospero File System: A Global File System Based on the Virtual System Model. *Computing Systems*, *5* (4). 407-432. Fall 1992.
23. Pearlman, L., Welch, V., Foster, I., Kesselman, C. and Tuecke, S. A Community Authorization Service for Group Collaboration. Globus Project, 2002.
24. Raman, S. and McCanne, S. A Model, Analysis, and Protocol Framework for Soft State-based Communication. *Computer Communication Review*, *29* (4). 1999.
25. Ratnasamy, S., Francis, P., Handley, M., Karp, R. and Shenker, S., A Scalable Content-Addressable Network. In *SIGCOMM Conference*, (2001), ACM
26. Ripeanu, M., Foster, I. and Iamnitchi, A. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. University of Chicago, 2001.
27. Rosenberg, J., Schulzrinne, H. and Suter, B. Wide Area Network Service Location. IETF, Internet Draft, 1997.
28. Ross, K.W. Hash routing for collections of shared Web caches. *IEEE Network*. 37-44. 1997.
29. Sharma, P., Estrin, D., Floyd, S. and Jacobson, V., Scalable Timers for Soft State Protocols. In *IEEE Infocom '97*, (1997), IEEE Press

30. Stockinger, H., Samar, A., Allcock, W., Foster, I., Holtman, K. and Tierney, B., File and Object Replication in Data Grids. In *10th IEEE Intl. Symp. on High Performance Distributed Computing*, (2001), IEEE Press, 76-86
31. Stoica, I., Morris, R., Karger, D., Kaashoek, F. and Balakrishnan, H., Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM Conference*, (2001), ACM
32. Vahdat, A., Eastham, P. and Anderson, T. WebFS: A Global Cache Coherent Filesystem. 1996.
33. Valloppillil, V. and Ross, K.W., Cache array routing protocol v1.0. In *Internet Draft*, (1988)
34. van Steen, M., Hauck, F., Homburg, P. and Tanenbaum, A. Locating Objects in Wide-Area Systems. *IEEE Communications Magazine*. 104-109. 1998.
35. Wang, J., A Survey of Web Caching Schemes for the Internet. In *Proceedings of ACM SIGCOMM '99 Conference*, (1999)
36. Wolman, A., Voelker, G.M., Sharma, N., Cardwell, N., Karlin, A. and Levy, H.M., On the scale and performance of cooperative Web proxy caching. In *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOPS'99)*, (Kiawah Island Resort, SC, USA, 1999). http://www.cs.washington.edu/research/networking/websys/pubs/sosp99/
37. Yu, P.S. and MacNair, E.A., Performance study of a collaborative method for hierarchical caching in proxy servers. In *Proceedings of 7th International World Wide Web Conference (WWW7)*, (1998)
38. Zhang, L., Braden, B., Estrin, D., Herzog, S. and Jamin, S., RSVP: A new Resource ReSerVation Protocol. In *IEEE Network*, (1993), 8-18
39. Zhao, B.Y., Kubiatowicz, J.D. and Joseph, A.D. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. UC Berkeley, Technical Report CSD-01-1141, 2001.